A Software Architecture Based on Coarse-Grained Self-Adjusting Computations

Stefan Wehr
stefan.wehr@hs-offenburg.de
University of Applied Sciences, Offenburg, Germany

Abstract

Ensuring that software applications present their users the most recent version of data is not trivial. Self-adjusting computations are a technique for automatically and efficiently recomputing output data whenever some input changes.

This article describes the software architecture of a large, commercial software system built around a framework for coarse-grained self-adjusting computations in Haskell. It discusses advantages and disadvantages based on longtime experience. The article also presents a demo of the system and explains the API of the framework.

CCS Concepts: • **Software and its engineering → Functional languages**.

Keywords: software architecture, self-adjusting computations, functional reactive programming, push, pull, Haskell

ACM Reference Format:
Stefan Wehr. 2023. A Software Architecture Based on Coarse-Grained Self-Adjusting Computations. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Functional Software Architecture (FUNARCH '23), September 8, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3609025.3609481

1 Introduction

Many software applications need to present their users the most recent version of data available. Further, data updates should propagate in a timely manner. Ensuring these two properties is not trivial, especially if input data is collected and aggregated from multiple sources that change over time.

There are two main approaches to deal with this challenge. With the *pull* approach, new versions of outputs are explicitly requested, e.g. by hitting a reload button or through client-side polling. In contrast, the *push* approach shifts responsibility to the input data sources and the processing pipeline. Each time a source emits new data, the change must be propagated to the outputs.

The pull approach is rather straightforward to implement. For each request, the software queries the data sources for inputs and computes the outputs. By construction, the output is up-to-date with respect to the inputs at request time. But if the request time is too far in the past, users may experience latency. Another downside is that requests are often performed redundantly for the same inputs.

The push approach avoids latency and such redundant computations because it recomputes outputs automatically on change of relevant inputs. Further, applications might benefit from precomputed outputs, e.g. to synchronize mobile devices for offline use. However, it is not trivial to ensure that outputs are up-to-date. On change of some input, all outputs depending on it must be recomputed (correctness), but other outputs should not (efficiency). Hence, developers need to track dependencies between inputs and outputs, either manually or, better, by relying on some framework.

This article describes a software architecture based on the push approach. Central to the architecture is a framework for *coarse-grained self-adjusting computations* [1, 20]. These computations ensure that any change to some input is propagated through the processing pipeline, ultimately keeping all outputs up-to-date. A dynamic dependence graph and memoization enables the framework to reevaluate only those parts of the pipeline that really depend on the changed input. Currently, the dependence graph is not persisted, it must be rebuilt after each start. This might delay application startup.

The framework provides abstractions for data sources and sinks to integrate external systems. These systems may use push, pull, or other approaches.

Contributions and Roadmap.

- We describe the architecture of a large, commercial software system built around a Haskell framework for coarse-grained self-adjusting computations (§ 3).
- We explain the API of the framework, which we often just call the *computation framework* (§ 4).
- We present a simplified demo of the original software (§ 5).
- We discuss advantages and disadvantages based on longtime experience implementing, extending, operating and maintaining the system (§ 6).

FUNARCH '23, September 8, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0297-6/23/09…$15.00
https://doi.org/10.1145/3609025.3609481

We start in § 2 with background information, § 7 discusses related work, and § 8 summarizes. The source code of the demo and the computation framework is available open-source [23]. We extracted the code from the original software, some parts have been written from scratch.

2 Background

The architecture and implementation described in this article is part of a large commercial software framework developed by medilyse GmbH (Freiburg, Germany). The author of this article worked 2010–2020 for this company. During that time, he was heavily involved in the architecture, the design, and the implementation of the whole system. Starting in 2010, medilyse had three employees, in 2023 there are about 15.

The software developed by medilyse, called *computation framework*, provides the technical foundation of *C*, a health information system used in several of the largest hospitals in Germany.[1] Main users of the system are doctors, nurses, and patients. They access *C* either through a web-browser or a mobile device. The mobile app supports offline operation to compensate bad WLAN coverage in some hospitals.

C offers a range of diverse features. (1) It provides doctors and nurses read-only access to the electronic health record (EHR) of a patient. (2) It gives doctors and nurses write-access to some data fields of the EHR. (3) It enables collaboration among doctors and nurses. (4) It allows patients to communicate with the hospital.

Feature (1) involves acquisition and aggregation of data from various sources provided by the hospital IT. This data includes personal information of patients, diagnoses, medications, treatment plans, radiology images, laboratory tests, surgery reports, and more. Features (2) – (4) are interactive features, subsumed under the name *workflow*.

This article mainly concentrates on the architecture supporting data processing for feature (1). Hospitals in Germany usually do not offer a central place for all information of the EHR. Instead, data is scattered over different systems, and each system has its own way of querying and retrieving data. Here are some of the main data sources:

- HL7 version 2 [13] communicates admissions, discharges, and transfers of patients, as well as various kinds of reports and test data. Nearly every hospital has its own in-house standard for the semantics of HL7 messages.
- DICOM [7] is a standard to transmit, store, and retrieve medical images and their meta data.
- There are various other systems (often hospital-specific) for accessing information such as archive data, surgery schedules, results of microbiological tests, and more.

To conclude this background section, here are approximate numbers for a large installation of *C*. The system processes 140,000 HL7 messages per day, the working set requires 1.6 TB of disk space and consists of 10,000 patients, 230,000 reports, 6.5 millions dicom images, and 500 users. *C* is implemented in a variety of programming languages: Haskell (375,000 lines of code), Typescript (330,000), Objective-C (150,000), and Dart (70,000).

[1] For legal reasons, we do not use the real name of the system and omit several brand, company, and hospital names.

3 System Architecture

Fig. 1 shows the architecture of *C* with a focus on how data flows through the system. The *hospital* part on the left is external to the system and depicts the other software systems of the hospital's IT infrastructure. Self-adjusting computations, shown as bold arrows in **red**, drive the processing pipeline. These computations automatically recompute outputs on change of some input, caching intermediate results. Rounded rectangles are software components, cylinders event stores, and dashed arrows represent data flowing in a specific direction. The processing pipeline has four main steps.

3.1 Import

The first step imports data from IT systems of the hospital. It features several import components, depicted as rounded rectangles in **green**, to integrate event-driven (push) and query-based (pull) hospital systems into the push-based pipeline of *C*. Two examples will clarify this idea.

HL7 is event-based. The **HL7 server** is configured in the hospital's communication server to receive HL7 messages for all state changes such as patient admission, discharge, transfer or the creation/update of a medical report. On reception of a message, the HL7 server saves the message and pushes a notification to the computation pipeline. In contrast, DICOM is query based. Hence, the **DICOM import** queries the hospital's DICOM server for new radiology images in regular intervals. On arrival of new images, it saves them to disk and pushes a notification to the computation pipeline.

3.2 Interpretation

Each hospital has its own conventions and standards. It is the responsibility of the interpretation to transform data coming from hospital IT systems into a data format common to all installations of *C*. Self-adjusting computations drive this transformations. They read input from import components (and thus ultimately from the hospital) and produce data in *C*'s **domain model** of the hospital.

For example, a lab value for C-reactive protein (CRP, an inflammation indicator) might be encoded in HL7 like this:

```
OBX|2|NM|CRP^^^103440|9989|0.9|mg/dL|< 0.5|H|||F|
```

Here is the representation in the domain model (JSON-format):

```
{"name": "CRP", "displayRange": "< 0.5",
 "displayUnit": "mg/dL", "displayValue": "0,9",
 "rangeMax": 0.5, "value": 0.9, "flag": "too-high"}
```

The interpretation persists its output (the domain model) in a component called *file store* (left cylinder filled with **blue**,

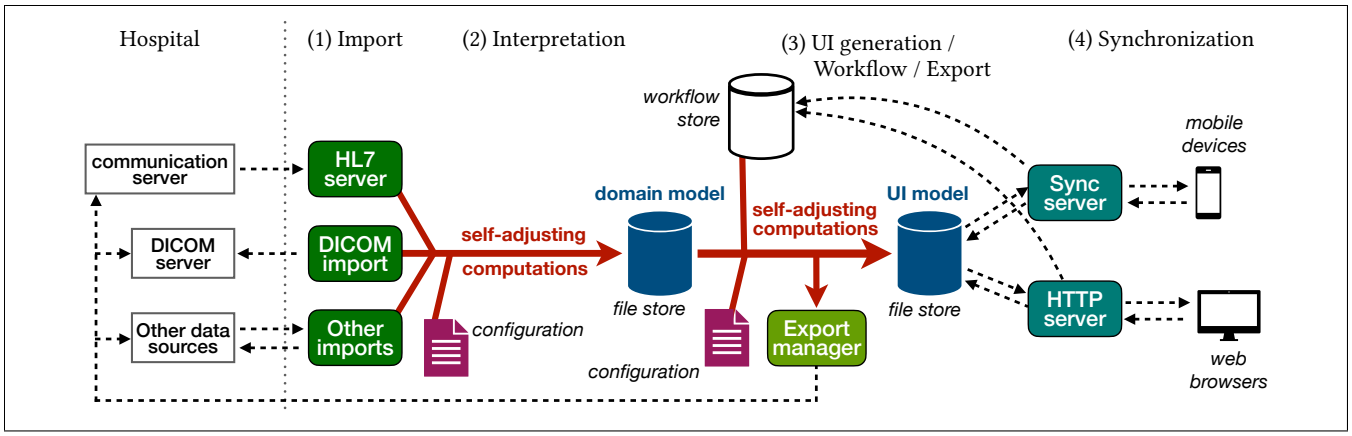

Figure 1. Architecture of *C* with a focus on data flow.

Fig. 1). The file store is a custom-made document store supporting multiple versions per document, garbage collection for unreachable documents, and an eventlog announcing changes to the documents. Essentially, it is an event store [18] that deletes unreachable data after a certain time. Deletion needs to be performed for reasons of data privacy and to keep disk usage manageable, as the working set without historic data already requires up to 1.6 TB on disk.

The processing pipeline uses another instance of the file store in the next step (right cylinder filled with **blue**). For historic reasons, the two instances of the file store do not share the same code. As they are very similar in interface and semantics, we ignore the differences for this article.

3.3 UI generation / Workflow / Export

The main duty of the third step is to generate a high-level UI representation that can be rendered by mobile devices and web browsers. Again, self-adjusting computations drive the translation from data in the hospital **domain model** (left file store) to the **UI model** (right file store). The CRP example from step (2) might be represented in the UI model like this:

```
{ "key": "CRP", "keyDetails": ["< 0.5"],
  "value": { "text": "0,9", "color": "red" },
  "valueDetails": ["mg/dL"] }
```

The rendering could look like this:

CRP	0,9
< 5	mg/dL

UI generation is heavily customized by configuration files. The configuration defines what views should be generated, how these views should look (ordering, colors), and more. Different user groups might have different view configurations. However, there is no more hospital-specific code.

The third step has two more responsibilities. Although they are important for the overall system, we only cover them briefly because they are not central to this article.

Workflow is the umbrella term for any user input and interactivity. Examples are: user marks a task as completed or user saves a questionnaire. The system encodes such interactions as *workflow actions*. Techniques from operational transform [10] enable actions to be executed concurrently, even when offline. The workflow engine tries to solve arising conflicts automatically. If this fails, it requires user intervention. An event store (workflow store, unfilled cylinder) persists the actions and makes them available to the computations.

Export transfers certain data back to the hospital. For example, a completed questionnaire might be stored as a PDF in the database of the **export manager**. The export manager then exports the PDF to the hospital.

3.4 Synchronization

The last step is synchronization. For mobile devices, which support offline mode, the **sync server** uses the UI model from the file store to compute the set of documents reachable for a certain user. Further, it tracks the set of documents stored on each device. On connection of a user, it synchronizes the missing documents to the device. For users connecting via web browsers, no synchronization is performed. Instead, a **HTTP server** sends the documents when needed.

4 API for Computations

The architecture described in the preceding section is mainly realized in Haskell [12]. Especially, the framework for coarse-grained self-adjusting computations (often simply called *computations*) has been implemented in this language. Fig. 2 lists the essential functions provided by the API of the framework. The API has four major entities:

Computation definitions specify the implementations of computations. *Type:* `CompDef p r`, where `p` is the parameter and `r` the result type of the computation.

Computation values represent computations at runtime. These values result from wiring computation definitions with previously defined computation values. *Type:* `Comp p r`.

Caching strategies determine if/how computation results are cached. *Type:* `CompCacheBehavior r`.

Data sources and sinks abstract over inputs and outputs. *Type classes:* `CompSrc` and `CompSink`.

```
-- Accessing data sources and sinks
compSrcReq     :: CompSrc s => TypedCompSrcId s -> CompSrcReq s a -> CompM a
compSinkReq    :: CompSink s => TypedCompSinkId s -> CompSinkReq s a -> CompM a
-- Defining computations
defineComp     :: (IsCompParam p, IsCompResult r)
               => String -> CompCacheBehavior r -> (p -> CompM r) -> CompDef p r
defineIncComp  :: (IsCompParam p, IsCompResult r, LargeHashable r)
               => String -> r -> (p -> r -> CompM r) -> CompDef p r
-- Calling computation values
evalComp       :: (IsCompParam p, IsCompResult r) => Comp p r -> p -> CompM (Maybe r)
evalCompOrFail :: (IsCompParam p, IsCompResult r) => Comp p r -> p -> CompM r
-- Wiring computation definitions
wireComp       :: (IsCompParam p, IsCompResult r) => CompDef p r -> CompWireM (Comp p r)
wireRecComp    :: (IsCompResult r, IsCompParam p) => (Comp p r -> CompDef p r) -> CompWireM (Comp p r)
compDriver     :: (IsCompParam p, IsCompResult r)
               => (CompFlowRegistry -> IO () -> IO ()) -> CompWireM (Comp p r) -> p -> IO ()
-- Caching computation result
fullCaching    :: (LargeHashable r, Show r) => CompCacheBehavior r
hashCaching    :: (LargeHashable r, Show r) => CompCacheBehavior r
-- Constraints
type IsCompParam p = (Show p, Typeable p, LargeHashable p)    type IsCompResult r = (Show r, Typeable r)
```

Figure 2. API for self-adjusting computations

Two other types are also important: `CompM` is the monad for implementing computations, and `CompWireM` is the monad for wiring computation definitions and values.

4.1 Accessing Data Sources and Sinks

Function `compSrcReq` takes the identifier of a data source and a request, and returns input according to the request. This function lives in the `CompM` monad because it is used to implement computations. The handling of outputs is similar: `compSinkReq` takes the identifier of some sink and a request, where the request contains the output to be produced. The return value of `compSinkReq` contains information about the completed request (e.g. a document store returns the version of the newly stored document). Users of the framework may provide their own data sources and sinks; see App. A.

4.2 Defining Computations

Function `defineComp` creates a computation definition. It takes a unique name for the computation, a caching strategy for the result, and the body of type `p -> CompM r`.

Example

Consider a computation returning the number of lines in some file.

```
numberOfLinesCompDef :: CompDef FilePath Int
numberOfLinesCompDef =
 defineComp "numberOfLines" fullCaching $ \p -> do
  string <- compSrcReq fileSrc (ReadTextFile p)
  pure (length (lines string))
```

Here, we assume a data source for reading files with identifier `fileSrc` and request `ReadTextFile`. The request returns the content of the file as a string. We discuss the caching strategy `fullCaching` at the end of this section. The computation is reevaluated if the file at path `p` changes on disk.

End example

Function `defineIncComp` is a more general version of `defineComp`. It defines an explicitly incremental computation that folds over the history of computation results. Hence, there is also an initial value `r` and the body of the computation gets access to the previous result.

4.3 Calling Computation Values

The body of a computation definition may call a computation value supplying some argument, see `evalComp`. The result is `Nothing` if the call fails; `evalCompOrFail` propagates failure to the caller.

Example

The following computation definition takes a computation value `c`, applies it to all filenames read from another file, and finally sums the results.

```
sumCompDef :: Comp FilePath Int
           -> CompDef FilePath Int
sumCompDef c =
 defineComp "sum" fullCaching $ \p -> do
  string <- compSrcReq fileSrc (ReadTextFile p)
  list <- mapM (evalCompOrFail c) (lines string)
  pure (sum list)
```

The computation is reevaluated if either the file at path `p` changes on disk, or any of the `evalCompOrFail` calls return a different result.

End example

4.4 Tracking Dependencies

A call of `evalComp`, `evalCompOrFail`, or `compSrcReq` records a dependency between the caller and the callee. More specifically, assume that we execute the body of some computation definition c applied to argument p (written $c\ p$).

- If execution hits `evalComp` $c'\ p'$, a dependency between $c\ p$ and $c'\ p'$ is recorded (analogously for `evalCompOrFail`).
- If execution hits `compSrcReq` $s\ q$ for data source s and request q, a dependency between $c\ p$ and (s, q) is recorded.

The dependence graph is dynamic. At start of executing $c\ p$, the old dependencies are deleted and freshly collected.

Whenever the result of $c'\ p'$ or (s, q) changes, application $c\ p$ is automatically reevaluated. If reevaluation produces a different result than before, callers of $c\ p$ are reevaluated and so on. However, if reevaluation of $c\ p$ produces the same result, no changes need to be propagated.

The framework also tracks outputs being produced via `compSinkReq` while running $c\ p$. If computation c is no longer applied to p, the outputs get deleted.

The framework identifies a computation application $c\ p$ by the name specified in the definition of c and by a hash-value of p. Hashing is performed via the type class `LargeHashable`, see constraint `IsCompParam`. This class provides a method `largeHash` for computing 128 or 256 bit cryptographic hashes. We have `largeHash`(x) == `largeHash`(y) iff x == y with a very high probability, so the framework may use `largeHash` to identify values.

4.5 Wiring Computation Definitions

Before use, a computation definition has to be turned into a computation value, see `wireComp`. Recursive computations are possible via `wireRecComp`. Both functions run in the `CompWireM` monad, which collects all computation values.

Function `compDriver` runs the collected computation values. The first parameter of this function is a callback taking a `CompFlowRegistry`. Users of the framework register all required data sources and sinks here.

Example

We complete the example by a computation that stores the result of a computation application in file `output.txt`.

```
storeCompDef :: Comp FilePath Int -> CompDef () ()
storeCompDef c =
 defineComp "store" fullCaching $ \() -> do
  i <- evalCompOrFail c "file_list.txt"
  compSinkReq fileSink (WriteTextFile "output.txt"
                        ("lines: " ++ show i))
```

Here, `fileSink` is the identifier of a data sink for writing files, `WriteTextFile` is a request of this sink. Next, we wire all three computations.

```
wireAllComps :: CompWireM (Comp () ())
wireAllComps = do
 numberOfLinesC <- wireComp numberOfLinesCompDef
 sumC <- wireComp (sumCompDef numberOfLinesC)
 wireComp (storeCompDef sumC)
```

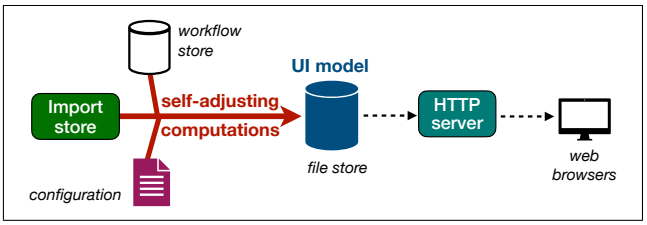

Figure 3. Architecture of the demo system

The main function (not shown) uses `compDriver` from Fig. 2 to run the computation value returned by `wireAllComps`.

End example

4.6 Caching Computation Results

The framework caches computation results. There are two main strategies:

- `fullCaching`: The result of calling a computation value is memoized. If the computation value is applied to the same argument elsewhere and none of its dependencies has changed, then the result is fetched from the cache. Typically, this strategy is used if the computation value is called multiple times with the same argument, if a call is not cheap, and if the result does not consume too much memory.
- `hashCaching`: The result of calling a computation value is not saved, only its large hash. Hence, if the computation value is applied to the same argument, it has to be reevaluated. Typically, this strategy is used when we need a dependency blocker, but caching the whole result would consume too much memory.
  Assume a call $c_1\ p_1$ evaluates $c_2\ p_2$ which in turn evaluates $c_3\ p_3$. Using `hashCaching` allows the framework to block the transitive dependency of $c_1\ p_1$ on $c_3\ p_3$, so $c_1\ p_1$ has to be reevaluated only if $c_2\ p_2$ changes but not if only $c_3\ p_3$ does. This is beneficial if $c_3\ p_3$ changes much more frequently than $c_2\ p_2$.

5 Computations in Action

This section makes the architecture and the API from § 3 and § 4 concrete. It gives a tiny, stripped-down demo of *C*.

Fig. 3 shows the architecture of the demo. The architecture is close in spirit to that of *C* but we made several simplifications. Hospital data is not imported as HL7, but expected to be stored as JSON in an event store (**Import store**). There is no dedicated domain model. Instead, we use the schema of the JSON data in the import store. Consequently, the "interpretation" is missing. We dropped support for exports and for mobile devices. Further, users cannot interact with the system by creating workflow actions. Instead, data has to be inserted manually as JSON into the workflow store.

Fig. 4 shows code from the demo, including some auxiliaries. The definition `getCfgCompDef` defines a computation

Computations

```
getCfgCompDef :: CompDef () Config
getCfgCompDef =
 defineComp "getCfg" fullCaching $ \() -> do
  bs <- compSrcReq fileSrc (ReadFile "demo.cfg")
  parseCfg bs

visiblePatsCompDef :: Comp () Config
   -> Comp () PatsAcc -> CompDef () PatMap
visiblePatsCompDef cfgC activePatsC =
 defineComp "visible" fullCaching $ \() -> do
  cfg <- evalCompOrFail cfgC ()
  (_, m) <- evalCompOrFail activePatsC ()
  now <- compGetTime TimeInterval5min
  pure (HashMap.filter (pred now cfg) m)
 where pred :: UTCTime -> Config -> Pat -> Bool
```

```
activePatsCompDef :: CompDef () PatsAcc
activePatsCompDef =
 defineIncComp "active" (Nothing,HashMap.empty) f
 where f :: () -> PatsAcc -> CompM PatsAcc
       -- read more patients from eventlog
       -- update PatMap accordingly

wireComps :: CompWireM (Comp () ())
wireComps = do
 cfgC <- wireComp getCfgCompDef
 activePatsC <- wireComp activePatsCompDef
 visiblePatsC <- wireComp $
   visiblePatsCompDef cfgC activePatsC
 ...
```

Auxiliaries

```
type PatMap = HashMap PatId Pat
type PatsAcc = (Maybe PatMsgKey, PatMap)
compGetTime :: TimeIntervalType -> CompM UTCTime
```

```
fileSrc :: TypedCompSrcId FileSrc
ReadFile :: FilePath -> CompSrcReq FileSrc ByteString
parseCfg :: MonadFail m => ByteString -> m Config
```

Figure 4. Code from the demo

for reading the configuration from file `demo.cfg`. The computation takes no parameter, as we hardcode the name of the config file. The `fullCaching` strategy means that the config file is only parsed on first invocation of the computation or if the file content changes. We read the file with `compSrcReq`. Here, `fileSrc` is the identifier of a `FileSrc`, a `CompSrc` instance for reading files. `ReadFile` is a request of this source returning a `ByteString`.

The definition `activePatsCompDef` computes the active patients from the import store. This store is an eventlog of patient messages, where each message carries an increasing key `PatMsgKey` and the data of a patient `Pat`. A patient has a patient ID `PatId`, some personal data, the admission time, and an optional discharge time. Newer messages update the data of older messages for the same patient.

Hence, `activePatsCompDef` uses `mkIncComDef` to fold over the messages from the store. The accumulator of type `PatsAcc` carries the key of the last message read from the store so far (optional, type `Maybe PatMsgKey`) and a map of currently active patients (type `PatMap`). The function `f`, whose body has been omitted, gets the previous `PatsAcc` value, reads new messages from the store, and computes the new accumulator. Note that `activePatsCompDef` cannot read old messages again. Thus, its decision to add or remove a patient from the active patients must be final for a specific message, it must not depend on the configuration. Consequently, `activePatsCompDef` over-approximates the set of patients visible to the users.

The definition `visiblePatsCompDef` computes the actually visible patients. It gets the two previously defined computations to access the configuration and the active patients. It then retrieves the current time to filter out certain active patients according to the configuration. Accessing the time is handled by a builtin data source exposed via `compGetTime`. This function takes a time interval (1min, 5min, …) specifying the granularity of updates to the time.

The other computations of the demo do not introduce any new concepts. We show the wiring of computations in `wireComps`. This function ultimately returns the root computation (not shown in the code). The order to construct the computations via `wireComp` starts at the leaves of the static call tree. This way, all computations required to define some computations c are already defined when defining c. For example, `getCfgCompDef` and `activePatsCompDef` take no computation as parameters, so these are defined first. Then we use the resulting computations for `visiblePatsCompDef`. Consequently, mutually recursive computations are not supported (but recursive computations are possible)

The main function of the demo (not shown) uses `wireComps` to define the required computations. Before handing control to the main loop of the computation framework, it also needs to register all required data sources and sinks. Some standard data sources and sinks are predefined, but defining custom sources or sinks is possible, see App. A.

The demo is rather small, it consists of eight computations, two sources, and one sink. The original software has 570 computations; 20 thereof use `hashCaching`, the remaining ones use `fullCaching`. It contains 13 data sources and sinks.

6 Discussion

Development of the system presented in this article started in 2010 and the product is still alive. The push-centered architecture based on the framework for coarse-grained self-adjusting computations was there from the beginning. We extended the framework with several features over the years

(e.g. exports, demand-driven generation of outputs, client-side scripting), so we can testify enough flexibility.

Clearly, developing a software product for hospitals also involves aspects such as privacy, security, regulatory affairs, and deployment. This article concentrates on the computation framework and deliberately ignores these points.

6.1 Advantages and Disadvantages

We had several motivations for devising a push-based architecture. Firstly, bad WLAN coverage in 2010 required an offline mode for the mobile application. (WLAN coverage in German hospitals has improved since then but is still an issue.) But support for offline operations requires to compute all data upfront, so that data can be synchronized to mobile devices. Secondly, we had seen lot of very slow software systems in hospitals, presumably because of database contention caused by repeated polling. As update rates in hospitals are quite low (less than 5Hz, most of the time much lower), we expected the push approach to solve this problem as well. Thirdly, we wanted the system to scale with the number of users, at least if there is a fixed number of view configurations, so that the majority of users see the same data. Overall, all three expectations were met.

But there are also disadvantages. The main problem is long startup times. If the systems restarts (most of the time due to a software update, very rarely because of a crash), it can take up to two hours (!) to get everything operational again. The problem here is that the state of the computation framework is not persistent. Hence, the system regenerates every output from scratch, just to notice the result on disk is already up-to-date. During this time, users can still interact with the system, they can view and input information, but no new data flows through the pipeline. At one point, we experimented with persisting the state of the framework. However, our approach turned out to be too slow, so we did not investigate further.

There are also scalability issues in certain directions. For example, doubling the number of patients (i.e. inputs) potentially visible to some users also doubles (at least) the time required to process all changes. This holds even if nobody actually accesses the added patients. To mitigate this problem, we introduced the concept of focus. For a certain group of patients, data is only generated if a user clicks on the patient in the UI. Hence, data for such patients is not available offline before the first user tries to view it. Another scalability problem arises if many users have different view configurations. In this case, the system needs to produce outputs for many different view configurations.

To summarize the discussion on scalability: the push approach performs work for each input change and each output, whereas the pull approach performs work for every client-side request.

6.2 Haskell and Alternative Approaches

We used the functional programming language Haskell to implement the computation framework. Haskell's emphasis on purity is important because computation bodies should contain only side effects from the `CompM` monad. Other side effects could lead to unwanted behavior as these effects would be repeated whenever the computation is reevaluated. Further, free monads are essential to temporarily suspend one computation in order to evaluate another one.

Our implementation does not rely on laziness, though. In fact, the whole project uses GHC's `StrictData` flag, so the fields of all data types are strict by default. Only very few fields are explicitly lazy (and we could even avoid these). Having strict data types makes space leaks much less likely.

It seems possible to implement the framework in some other programming language. Using a more mainstream language (Kotlin, Dart, …) would offer better support for debugging and monitoring than Haskell. But we would miss monads and a type system that controls side effects.

The software product discussed in this article uses a variety of languages: Haskell for the backend, Typescript for client-side scripting and the web, Objective-C for the legacy iOS app, and Dart for the next-generation client app. This mixture of languages introduces its own complexity and becomes more of a problem, especially as the size of the team grows. (We started with three developers, now there are eight). For example, a developer proficient in Typescript but not in Haskell tends to solve problems in Typescript, even if the right solution lies in the Haskell part of the application. Also, Haskell's learning curve is quite steep, and a good part of the Haskell code is structured in a rather unusual way (Fig. 4). These points are not in favor of Haskell.

The computation framework was there from the beginning, initially only for the UI generation step (Fig. 1). We did not really try alternatives because everything worked well. The interpretation step was developed later. Based on our previous experience, we used the computation framework for this part.

7 Related Work

7.1 Incremental Computations

Incremental computations [20] is the technique of efficiently updating the result of a computation on change of the inputs. The dependence graph can be static [6] or dynamic, which leads to *adaptive computations* [2]. Acar and coworkers [1] combine adaptive computations with memoization to arrive at *self-adjusting computations*. They present their implementation as a library for Standard ML, relying on modifiable references to represent the dependencies between computations. This allows for fine-grained dependencies. For example, self-adjusting algorithms on lists react on deletions and insertions of individual list elements. Our framework also maintains a dynamic dependence graph, but this graph

records dependencies between computations. Hence, our approach is much more coarse-grained. There exists an OCaml implementation for self-adjusting computations [14].

7.2 Functional Reactive Programming

(Functional) reactive programming [9, FRP] is closely related to our approach. FRP offers two major abstractions: *behaviors* for continuously time-varying values and *events* for streams of discrete state changes. Our approach does not support behaviors, all state changes are discrete. A survey of Bainomugisha and colleagues [3] introduces six characteristics of reactive programming: basic abstractions (our approach: computations), evaluation model (push, but integration of pull-based sources is possible), support for glitch avoidance (no, see below), lifting operations (explicit), multidirectionality (no), distribution (no). Often, FRP serves to program graphical user interfaces, supporting events such as key strokes and mouse clicks. Our computation framework does not have this purpose.

Some implementations of FRP use a pull-based evaluation model, others rely on push. Elliott [8] uses push-pull. This hybrid approach evaluates continuous behaviors in pull-style. For discrete events, it uses push to avoid unnecessary recomputations and to optimize latency.

FrTime [5] is a reactive extension of Racket [21]. It uses push-based change propagation, supports higher-order behaviors/events, and lifts primitive operations automatically to behaviors/events. In contrast, our computations are first-order, and the `CompM` monad acts as type barrier between computations and pure expression, so explicit lifting is required. A glitch is an update inconsistency that occurs when change propagation causes some parts of the program to see the new values, whereas other parts still see the old values. FrTime avoids glitches by imposing a topological ordering on dependencies. With our approach, glitches are possible because computations might read new values from data sources before these values propagate at all.

Margara and Salvaneschi [15] examine change propagation for distributed reactive programming. While our system as a whole is distributed, the computation framework is not. Reynders and coworkers [22] combine reactive programming with multi-tier programming for writing web applications. Zhao and colleagues [24] present a Haskell implementation of a push-pull reactive programming model in the domain of IoT applications. In this domain, update rates can be very high (10KHz and more). In our hospital domain, updates rates are significantly lower (less than 5Hz, most of the time much lower). Reactive values and relations [19] allow programmers to connect values that change over time, while retaining control over change propagation. Similar to our abstractions for data sources and sinks, reactive values may integrate external entities such as files or GUI widgets.

7.3 Build Systems

Our approach has strong similarities with build systems. One important differences is that build systems usually store intermediate and final artifacts on disk, whereas our system stores intermediate results in memory (depending on the caching strategy) and only final artifacts on disk. In the terminology of Mokhov and coworkers [17, Table 1], our framework supports *dynamic dependencies* and the *early cutoff optimization*. Further, the scheduler is *suspending* and computations using `fullCaching` are *minimal*. Our framework works mainly with local data, but some data could also be stored remotely to be accessed from different instances of the system. Our system is continuously running, holding the dynamic dependence graph in memory without persisting the state on disk.

The build tool Shake [16] has been a source of inspiration in the implementation of our system. For example, we use a similar approach to encode parameters and results of computations (keys and values in Shake). Further, our system also uses different monads for implementing the body of a computation (`CompM`, in Shake `Action`) and for wiring computations (`CompWireM`, in Shake `Rule`).

7.4 Architectural Patterns

Pipes and filters is an architectural pattern for systems that process a stream of data [4]. Our system can be viewed as an instance of this pattern, where computations are the filters and calls of computations are the pipes. One tries to achieve loose coupling between filters. In our system, computations are often quite tightly coupled.

Event-driven architectures [11] use events to connect loosely coupled, often distributed components. Processing of events is typically done asynchronously. In contrast, the computations presented in this article are tightly coupled, local, and communicate via synchronous calls.

8 Summary

This article presented a push-based architecture that forms the foundation of a commercial hospital information system. The system has been used in German hospitals for more than ten years. The architecture is realized in Haskell, based on a framework for course-grained self-adjusting computations. An important topic for future work is support for persistence of the framework state, so that restarts become much faster.

Data availability

The source code for this article is available open-source [23].

Acknowledgments

This article would not have been possible without all the hard work from my former colleagues at medilyse GmbH. Further, I would like to thank the anonymous reviewers of FUNARCH '23 for their detailed and helpful comments.

A Sources and Sinks

The computation framework abstracts over data sources for inputs and data sinks for outputs. Fig. 5 shows the API.

A.1 Sources

A data source `s` is an instance of type class `CompSrc`. The associated type `CompSrcReq s a` is the request type of the source, where `a` is the result of the request. The computation framework tracks dependencies between computation applications and inputs. If the input changes, computation applications that previously read the input are reevaluated. The type `Dep k v` models such a dependency; `k` is a key identifying the input and `v` is its version. The associated types `CompSrcKey` and `CompSrcVer` specify the key and the version type of a specific source. Versions do not have to appear in some increasing order, they just have to be different if the input changes. A data source has to provide four methods:

- `compSrcInstanceId`: Identifier of a source instance.
- `compSrcExecute`: Executes a request and returns a set of dependencies and the result of the request (the type `Fail a` is a strict variant of `Either String a`). This method is used when a computation application invokes `compSrcReq`, see § 4. The set of dependencies specify which inputs were touched. This set is used internally by the framework to judge when to reevaluate the computation application.
- `compSrcWaitChanges`: Used by the framework to wait for changes to the inputs that `compSrcExecute` previously returned. The result is a set of dependencies that have changed.
- `compSrcUnregister`: Used by the framework to tell the source that it should no longer monitor the given keys. After that, `compSrcWaitChanges` no longer return changes for the given keys.

As an example, we consider the `FileSrc` already mentioned in § 5.

```
data FileSrc = FileSrc
  { src_ident :: CompSrcInstanceId
  , src_watch :: FileWatch }
data FileSrcReq a where
 ReadFile :: FilePath -> FileSrcReq ByteString
 ListDir ::
  FilePath -> FileSrcReq (HashSet DirEntry)
instance CompSrc FileSrc where
 type CompSrcReq FileSrc = FileSrcReq
 type CompSrcKey FileSrc = FilePath
 type CompSrcVer FileSrc = Maybe POSIXTime
```

The request type allows reading files and directories. (Type `DirEntry` contains the name and the kind of a directory entry.) The dependency key is the canonical path of the file/directory, the version is the optional modification time (nothing after deletion). The implementation of `compSrcExecute` reads the file or directory. Internally, the file source maintains a set of paths which it monitors for changes in the background (the `scr_watch` field). The implementation of `compSrcWaitChanges` returns the accumulated changes, and `compSrcUnregister` deletes paths from the set.

A.2 Sinks

A data sink `s` is an instance of type class `CompSink`. The associated type `CompSinkReq s a` is the request type of the sink, where `a` is the result of the request. If the request just produces some outputs, `a` is the unit type. The computation framework tracks the set of outputs generated by a computation application. If the computation application dies (i.e. the computation is no longer applied to the argument of the application), the outputs are deleted. The associated type `CompSinkOut s` is the output type of the sink. There are four methods to implement:

- `compSinkInstanceId`: Identifier of a sink instance.
- `compSinkExecute`: Executes a request and returns a set of outputs and the result of the request. This method is used when a computation application invokes `compSinkReq`, see § 4. The set of outputs specifies which outputs have been generated. It is used internally by the framework to delete outputs that are no longer generated.
- `compSinkDeleteOutputs`: Used by the framework to delete the given outputs.
- `compSinkListExistingOutputs`: Lists the existing outputs. The framework uses this method after a restart to identify the outputs that became garbage while the program was not running.

As an example, we consider `FileSink`, a sink for writing files and directories to some base directory.

```
data FileSink = FileSink
  { fcs_ident :: CompSinkInstanceId
  , fcs_root :: FilePath }
data FileSinkReq a where
 WriteFile ::
  FilePath -> ByteString -> FileSinkReq ()
 MakeDirs :: FilePath -> FileSinkReq ()
data FileSinkOut = FileSinkOut FilePath FileOrDir
instance CompSink FileSink where
 type CompSinkReq FileSink = FileSinkReq
 type CompSinkOut FileSink = FileSinkOut
```

The request allows writing files and creating directories. The output type contains the path and a flag specifying whether it is a file or a directory. The implementation of `compSinkExecute` writes the file or creates the directories, `compSinkDeleteOutputs` deletes the given files and directories from disk, and `compSinkListExistingOutputs` lists all files and directories starting at the root directory.

References

[1] Umut A. Acar, Guy E. Blelloch, Matthias Blume, and Kanat Tangwongsan. 2006. An experimental analysis of self-adjusting computation. In *Proc. of PLDI 2006*. ACM. https://doi.org/10.1145/1133981.1133993

Sources

```
class (Typeable s, IsCompFlowData (CompSrcKey s), IsCompFlowData (CompSrcVer s)) => CompSrc s where
  type CompSrcReq s  :: Type -> Type
  type CompSrcKey s  :: Type
  type CompSrcVer s  :: Type
  compSrcInstanceId  :: s -> CompSrcInstanceId
  compSrcExecute     :: s -> CompSrcReq s a -> IO (CompSrcDeps s, Fail a)
  compSrcWaitChanges :: s -> STM (CompSrcDeps s)
  compSrcUnregister  :: s -> HashSet (CompSrcKey s) -> IO ()

type IsCompFlowData a = (Show a, Eq a, Typeable a, Hashable a)
type CompSrcDep s = Dep (CompSrcKey s) (CompSrcVer s)
type CompSrcDeps s = HashSet (CompSrcDep s)
data Dep k v = Dep { dep_key :: k , dep_ver :: v }
  deriving (Eq, Ord, Data, Typeable, Generic, Hashable)
```

Sinks

```
class (Typeable s, IsCompFlowData (CompSinkOut s)) => CompSink s where
  type CompSinkReq s :: Type -> Type
  type CompSinkOut s :: Type
  compSinkInstanceId :: s -> CompSinkInstanceId
  compSinkExecute :: s -> CompSinkReq s a -> IO (CompSinkOuts s, Fail a)
  compSinkDeleteOutputs :: s -> CompSinkOuts s -> IO ()
  compSinkListExistingOutputs :: s -> Option (IO (CompSinkOuts s))

type CompSinkOuts s = HashSet (CompSinkOut s)
```

Figure 5. API for sources and sinks

[2] Umut A. Acar, Guy E. Blelloch, and Robert Harper. 2006. Adaptive functional programming. *ACM Trans. Program. Lang. Syst.* 28, 6 (2006). https://doi.org/10.1145/1186632.1186634

[3] Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. 2013. A survey on reactive programming. *ACM Comput. Surv.* 45, 4 (2013). https://doi.org/10.1145/2501654.2501666

[4] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. 2007. *Pattern-oriented software architecture, 4th Edition.* Wiley. https://www.worldcat.org/oclc/314792015

[5] Gregory H. Cooper and Shriram Krishnamurthi. 2006. Embedding Dynamic Dataflow in a Call-by-Value Language. In *Proc. of ESOP 2006 (LNCS, Vol. 3924).* Springer. https://doi.org/10.1007/11693024_20

[6] Alan Demers, Thomas Reps, and Tim Teitelbaum. 1981. Incremental Evaluation for Attribute Grammars with Application to Syntax-Directed Editors. In *Proc. of POPL 1981.* ACM. https://doi.org/10.1145/567532.567544

[7] DICOM 2023. DICOM standard. https://www.dicomstandard.org.

[8] Conal Elliott. 2009. Push-pull functional reactive programming. In *Proc. of Haskell Symposium 2009.* ACM. https://doi.org/10.1145/1596638.1596643

[9] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *Proc. of ICFP 1997.* ACM. https://doi.org/10.1145/258948.258973

[10] Clarence A. Ellis and Simon J. Gibbs. 1989. Concurrency Control in Groupware Systems. In *Proc. of SIGMOD 1989.* ACM. https://doi.org/10.1145/67544.66963

[11] Ludger Fiege, Gero Mühl, and Felix C. Gärtner. 2002. Modular event-based systems. *Knowl. Eng. Rev.* 17, 4 (2002). https://doi.org/10.1017/S0269888903000559

[12] Haskell 2023. The programming language Haskell. https://www.haskell.org.

[13] HL7 2023. HL7 standards. https://www.hl7.org.

[14] Incremental 2023. Incremental — Library for Incremental Computations. https://opensource.janestreet.com/incremental.

[15] Alessandro Margara and Guido Salvaneschi. 2018. On the Semantics of Distributed Reactive Programming: The Cost of Consistency. *IEEE Trans. Software Eng.* 44, 7 (2018). https://doi.org/10.1109/TSE.2018.2833109

[16] Neil Mitchell. 2012. Shake before building: replacing make with haskell. In *Proc. of ICFP 2012.* ACM. https://doi.org/10.1145/2364527.2364538

[17] Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. 2020. Build systems à la carte: Theory and practice. *J. Funct. Program.* 30 (2020). https://doi.org/10.1017/S0956796820000088

[18] Michiel Overeem, Marten Spoor, Slinger Jansen, and Sjaak Brinkkemper. 2021. An empirical characterization of event sourced systems and their schema evolution - Lessons from industry. *J. Syst. Softw.* 178 (2021). https://doi.org/10.1016/j.jss.2021.110970

[19] Ivan Perez and Henrik Nilsson. 2015. Bridging the GUI gap with reactive values and relations. In *Proc. of Haskell Symposium 2015.* ACM. https://doi.org/10.1145/2804302.2804316

[20] William W. Pugh and Tim Teitelbaum. 1989. Incremental Computation via Function Caching. In *Proc. of POPL 1989.* ACM. https://doi.org/10.1145/75277.75305

[21] Racket 2023. Racket, the Programming Language. https://racket-lang.org.

[22] Bob Reynders, Frank Piessens, and Dominique Devriese. 2020. Gavial: Programming the web with multi-tier FRP. *Art Sci. Eng. Program.* 4, 3 (2020). https://doi.org/10.22152/programming-journal.org/2020/4/6

[23] Stefan Wehr. 2023. Software repository for *A Software Architecture Based on Coarse-Grained Self-Adjusting Computations.* https://github.com/skogsbaer/computations and http://doi.org/10.5281/zenodo.8147256.

[24] Tian Zhao, Adam Berger, and Yonglun Li. 2020. Asynchronous monad for reactive IoT programming. In *Proc. of REBLS 2020.* ACM. https://doi.org/10.1145/3427763.3428314

Received 2023-06-01; accepted 2023-06-28